Zübeyir Bodur

21702382

CS202-1 HOMEWORK#1

Q1) Trace the algorithms for the array using:

A) Insertion Sort

The items in the array after the i-th iteration, number of key comparisons and swap operations in that iteration are shown below. Sorted part of the array is **black** and unsorted part is **bold**.

Initial:	**4**	**8**	**3**	**7**	**6**	**2**	**1**	**5**			
After 1st Iter:	**4**	**8**	**3**	**7**	**6**	**2**	**1**	**5**	#Key Comp..:	1	#Swap Op.: 0
After 2nd Iter:	**3**	**4**	**8**	**7**	**6**	**2**	**1**	**5**	#Key Comp..:	2	#Swap Op.: 2
After 3rd Iter:	**3**	**4**	**7**	**8**	**6**	**2**	**1**	**5**	#Key Comp..:	2	#Swap Op.: 1
After 4th Iter:	**3**	**4**	**6**	**7**	**8**	**2**	**1**	**5**	#Key Comp..:	3	#Swap Op.: 2
After 5th Iter:	**2**	**3**	**4**	**6**	**7**	**8**	**1**	**5**	#Key Comp..:	5	#Swap Op.: 5
After 6th Iter:	**1**	**2**	**3**	**4**	**6**	**7**	**8**	**5**	#Key Comp..:	6	#Swap Op.: 6
After 7th Iter:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp..:	4	#Swap Op.: 3

B) Selection Sort

The items in the array after the i-th swap and number of key comparisons to find the biggest item in the unsorted array (the black one) is shown below. Key comparisons are written before their swap operations because they are made before the swap operation. Sorted part of the array is **black**, the biggest element of the unsorted part is **red** and unsorted part is **bold**.

Initial:	**4**	**8**	**3**	**7**	**6**	**2**	**1**	**5**	#Key Comp.:	7
After 1st Swap:	**4**	**5**	**3**	**7**	**6**	**2**	**1**	**8**	#Key Comp.:	6
After 2nd Swap:	**4**	**5**	**3**	**1**	**6**	**2**	**7**	**8**	#Key Comp.:	5
After 3rd Swap:	**4**	**5**	**3**	**1**	**2**	**6**	**7**	**8**	#Key Comp.:	4
After 4th Swap:	**4**	**2**	**3**	**1**	**5**	**6**	**7**	**8**	#Key Comp.:	3
After 5th Swap:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	2
After 6th Swap:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	1
After 7th Swap:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	0

C) Bubble Sort

The items in the array after each iteration (the moment when the bubble have passed through every item in the unsorted part), # key comparisons and swap operations are shown below. Sorted part of the array is **green** unsorted part is **bold**.

Initial:	**4**	**8**	**3**	**7**	**6**	**2**	**1**	**5**				
1st Iter:	**4**	**3**	**7**	**6**	**2**	**1**	**5**	**8**	#Key Comp.:	7	#Swap Op.:	6
2nd Iter:	**3**	**4**	**6**	**2**	**1**	**5**	**7**	**8**	#Key Comp.:	6	#Swap Op.:	5
3rd Iter:	**3**	**4**	**2**	**1**	**5**	**6**	**7**	**8**	#Key Comp.:	5	#Swap Op.:	3
4th Iter:	**3**	**2**	**1**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	4	#Swap Op.:	2
5th Iter:	**2**	**1**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	3	#Swap Op.:	2
6th Iter:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	2	#Swap Op.:	1
7th Iter:	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	#Key Comp.:	1	#Swap Op.:	0

D) Merge Sort

The items in the array are partitioned into sub arrays, which are emphasized as groups. The steps where merge and mergeSort are called shown below.

Initial:	**4 8 3 7 6 2 1 5**	mergeSort {4,8,3,7,6,2,1,5}
Step 1:	**4 8 3 7** 6 2 1 5	mergeSort {4,8,3,7}
Step 2:	**4 8** 3 7 6 2 1 5	mergeSort {4,8}
Step 3:	**4** **8** 3 7 6 2 1 5	merge {4} and {8}
Step 4:	4 8 **3 7** 6 2 1 5	mergeSort {3,7}
Step 5:	4 8 **3** **7** 6 2 1 5	merge{3} and {7}
Step 6:	**4 8** **3 7** 6 2 1 5	merge {4,8} and {3,7}
Step 7:	**3 4 7 8** **6 2 1 5**	mergeSort {6,2,1,5}
Step 8:	3 4 8 7 **6 2** 1 5	mergeSort{6,2}
Step 8:	3 4 8 7 **6** **2** 1 5	merge {6} and {2}
Step 8:	3 4 8 7 2 6 **1 5**	mergeSort{1,5}
Step 8:	3 4 8 7 6 2 **1** **5**	merge{1,5}
Step 8:	3 4 8 7 **6 2** **1 5**	merge {6,2} and {1,5}
Step 9:	**3 4 8 7** **1 2 5 6**	merge {3,4,8,7} and {1,2,5,6}
Final Array:	1 2 3 4 5 6 7 8	

E) Quick Sort

First item will be chosen as pivot and the array will be divided into two parts, one of them being less than pivot and the other greater than or equal to pivot. The steps where partition and quickSort are called shown below. The partitions after the partition function is called are **emphasized**.

Initial:	**4 8 3 7 6 2 1 5**	pivot = 4, call partition
1st Partition:	**1 3 2 4 6 8 7 5**	quickSort call for {1, 3, 2}
	1 3 2 4 6 8 7 5	pivot = 1, call partition
2nd Partition:	**1 3 2 4 6 8 7 5**	quickSort for {} and {3, 2}
:	**1 3 2 4 6 8 7 5**	pivot = 3, call partition
3rd Partition:	**1 2 3 4 6 8 7 5**	quickSort for {2} and {6,8,7,5}
:	**1 2 3 4 6 8 7 5**	pivot = 6, call partition
4th Partition:	**1 2 3 4 5 6 7 8**	quickSort for {5} and {7,8}
:	**1 2 3 4 5 6 7 8**	pivot = 7, call partition
5th Partition:	**1 2 3 4 5 6 7 8**	quickSort for {} and {8}

Q2) Recurrence Eqn. for

A) Mergesort Algorithm

In the worts case, time requirement for merge function will be θ(n), and time requirement for a single item will be T(1)= θ(1). Then, we can write the time complexity for the recursive mergeSort function as the following.

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

The solution for this equation using repeated substitutions is written below.

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \theta(\frac{n}{2})\right] + \theta(n) = 4T\left(\frac{n}{4}\right) + 2\theta(\frac{n}{2}) + \theta(n) = 4T\left(\frac{n}{4}\right) + 2\theta(n)$$

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \theta(\frac{n}{4})\right] + 2\theta(n) = 8T\left(\frac{n}{8}\right) + 4\theta(\frac{n}{4}) + 2\theta(n) = 8T\left(\frac{n}{8}\right) + 3\theta(n)$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + k\theta(n)$$

Let k = $\log_2 n$. Then $2^k=1$. The equation becomes:

$$T(n) = T(1) + \log_2(n) \cdot \theta(n)$$
$$T(n) = \theta(n \cdot \log n)$$

Mergesort algorithm is θ(nlogn) in the worst case.

B) Quicksort Algorithm

The worst case in this algorithm is to get one of the partitions (S1 or S2) as empty array every single time, which happens when the array is already sorted and the pivot is chosen as the first item of the array. If this is the case, the array will be partitioned as {},{pivot},{S2} or vice versa. This means size of S2 will be always one less than the array. This can be written to the recurrence equation as the following:

$$T(n) = T(n-1) + T(0) + \theta(n)$$

We know that T(0) = T(1) = θ(1). Then, the equation is

$$T(n) = T(n-1) + \theta(n)$$

We can solve this using repeated substitutions:

$$T(n) = T(n-1) + \theta(n)$$
$$T(n) = T(n-2) + 2\theta(n)$$
$$T(n) = T(n-3) + 3\theta(n)$$
$$T(n) = T(n-k) + k\theta(n)$$

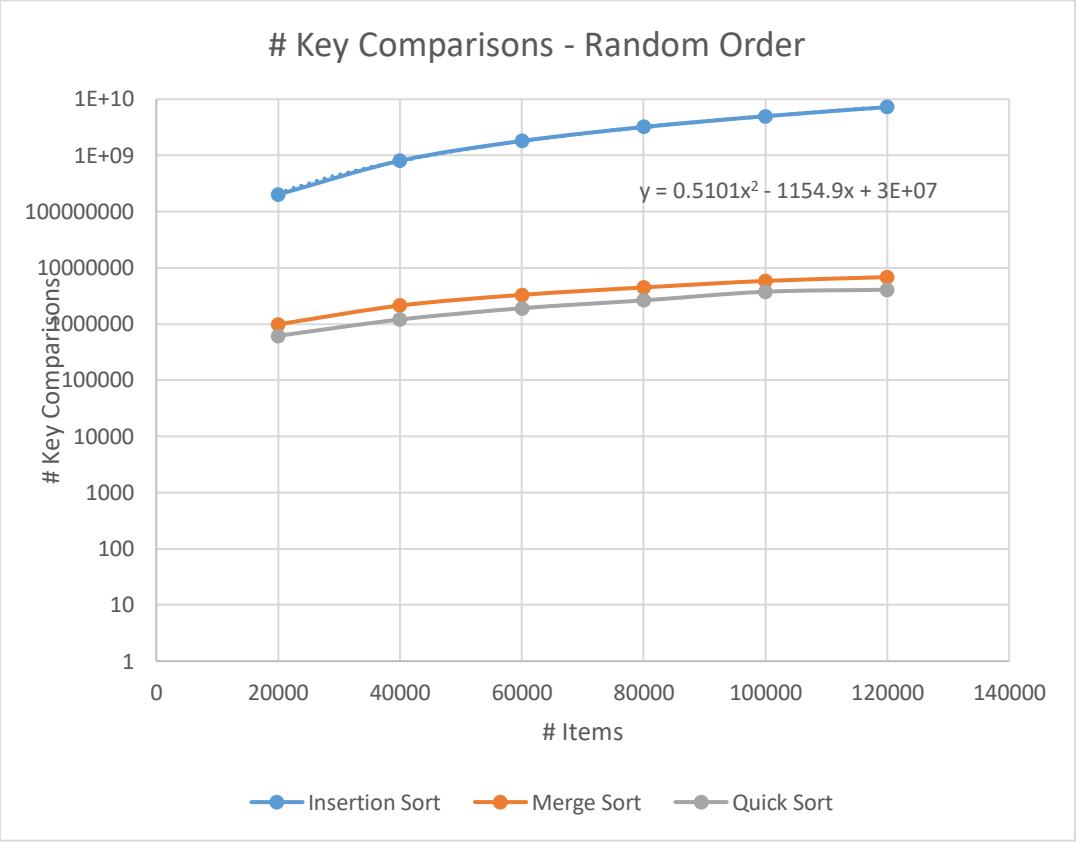
Let k = n - 1. Then n - k = 1.

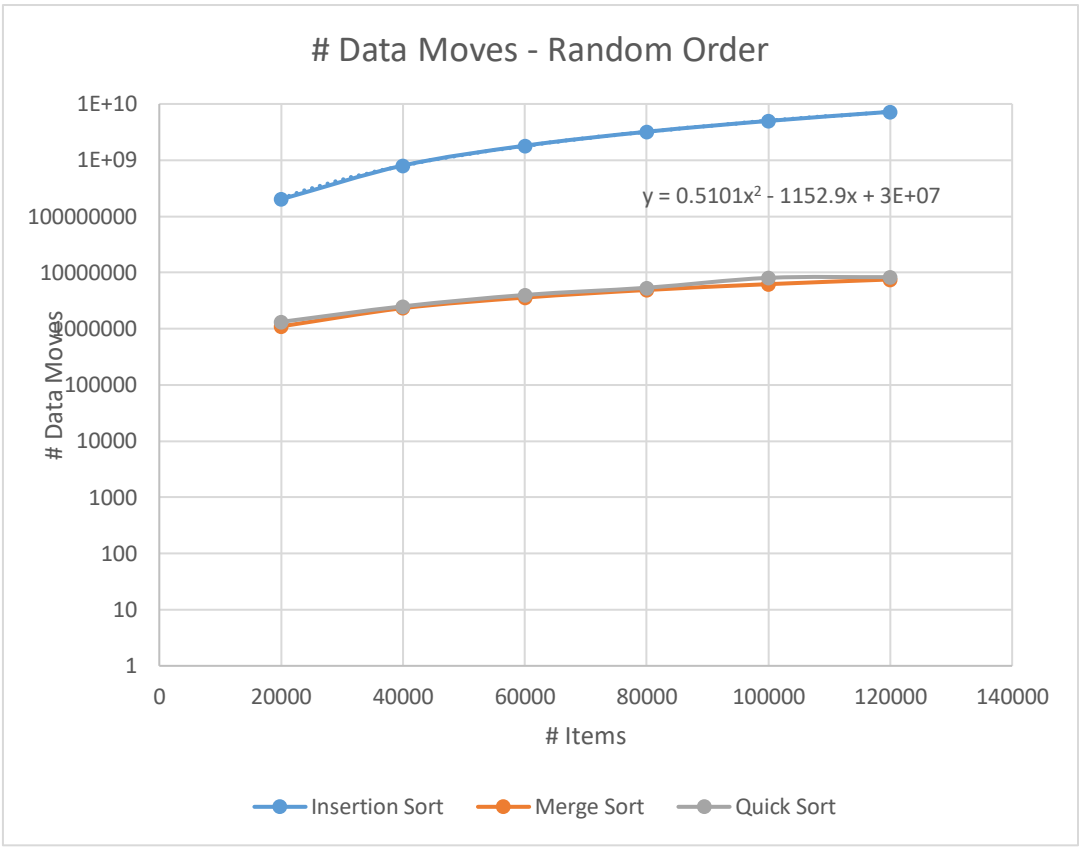
$$T(n) = T(1) + (n-1)\theta(n)$$
$$T(n) = \theta(n^2)$$

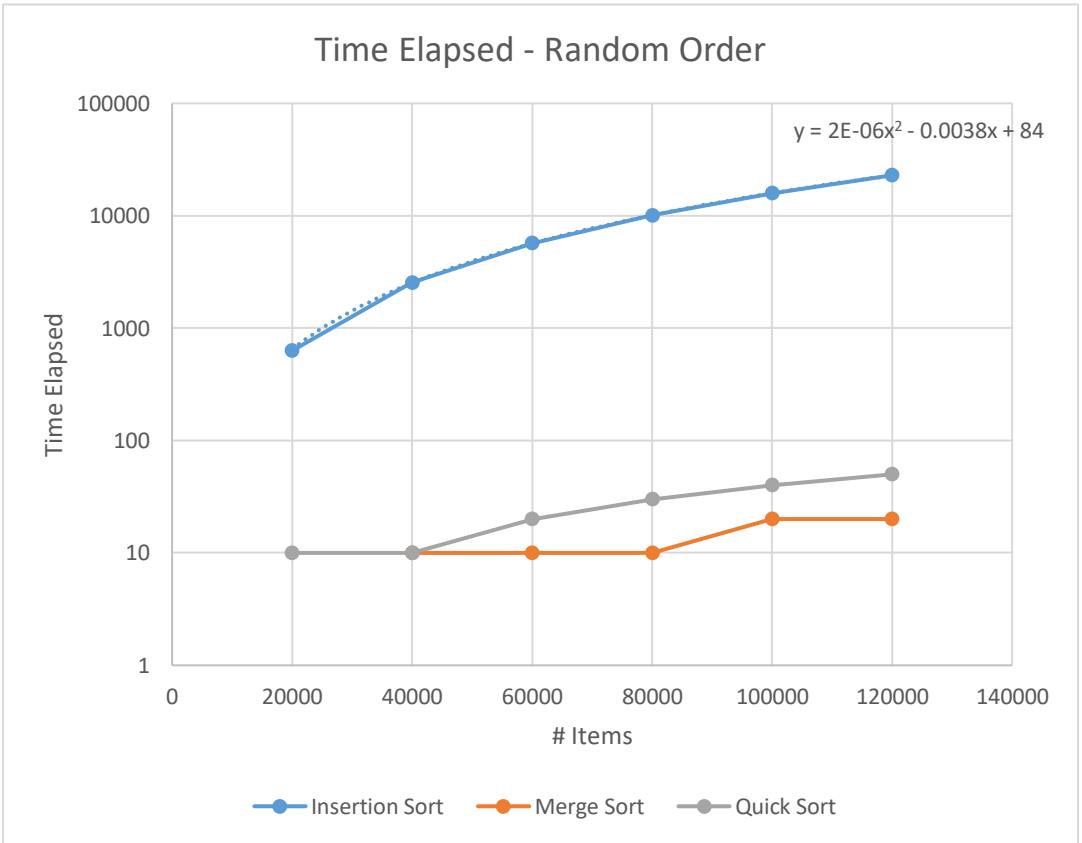
Quicksort algorithm is θ(n^2) in the worst case.

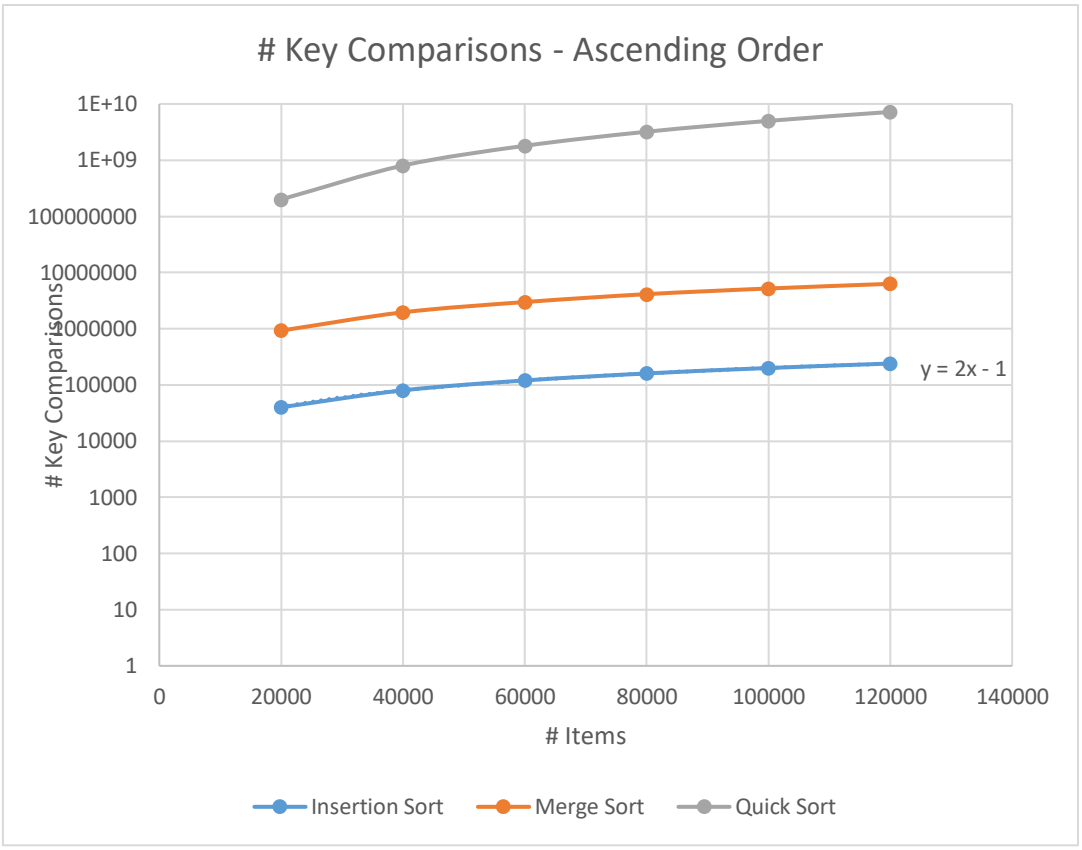
Q3) Experiment Results

Each plot for key comparisons, time elapsed and data moves depending on the order of the array is shown below. As some of the lines were not visible due to the gap between algorithms, the y-axis of the plots were shown logarithmically.


Key Comparisons - Random Order
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
# Key Comparisons
y = 0.5101x² - 1154.9x + 3E+07
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort

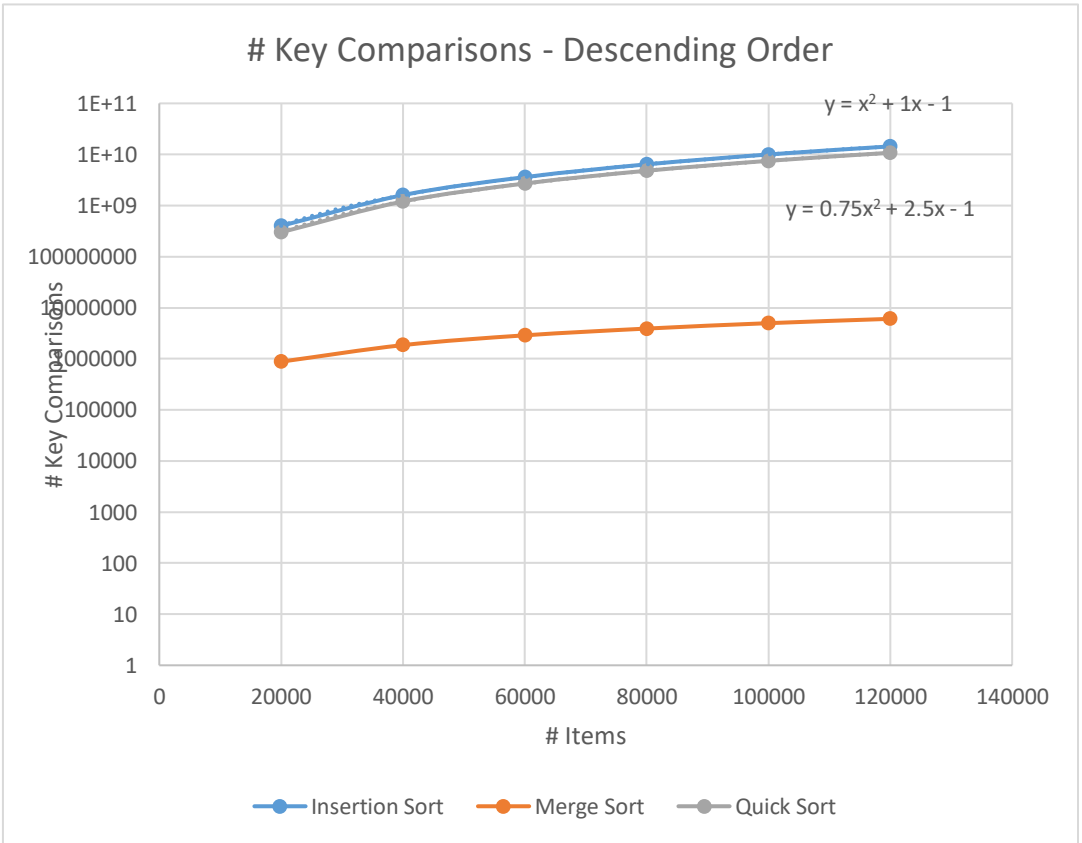

# Data Moves - Random Order
y = 0.5101x² - 1152.9x + 3E+07
# Data Moves
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort

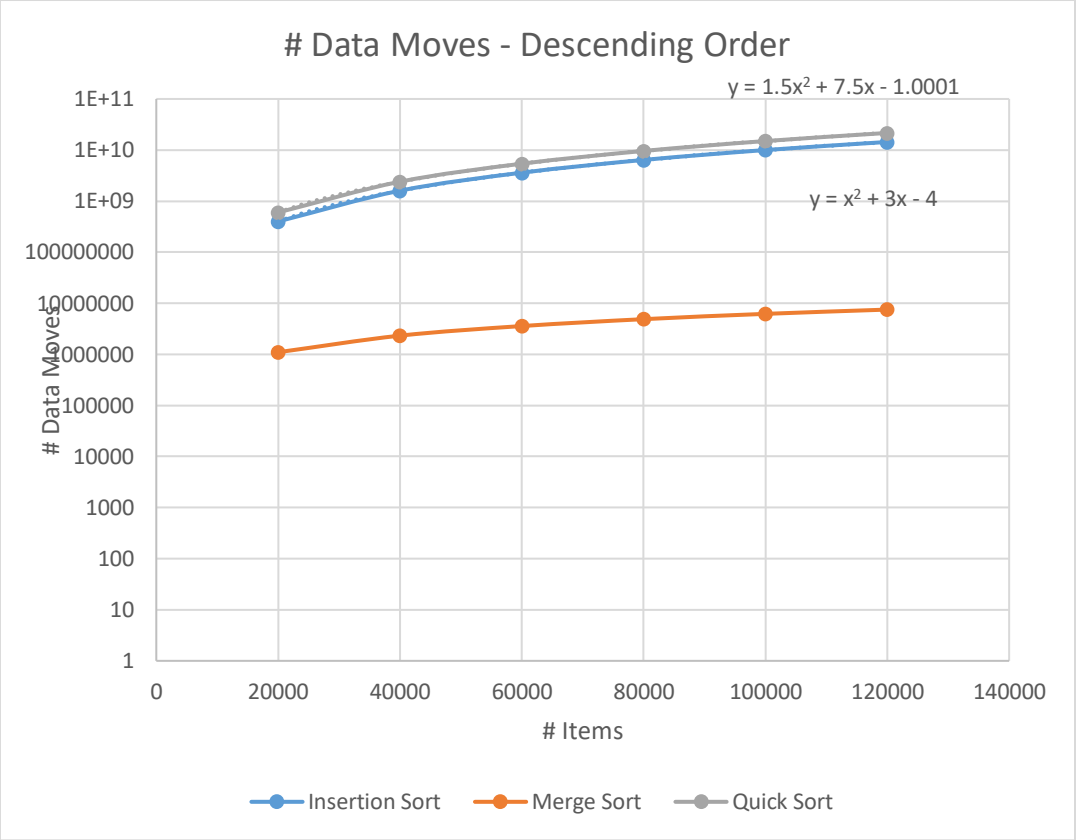

Time Elapsed - Random Order
y = 2E-06x² - 0.0038x + 84
Time Elapsed
100000
10000
1000
100
10
1
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort


Key Comparisons - Ascending Order
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Key Comparisons
0
20000
40000
60000
80000
100000
120000
140000
# Items
y = 2x - 1
Insertion Sort
Merge Sort
Quick Sort


Data Moves - Ascending Order
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Data Moves
0
20000
40000
60000
80000
100000
120000
140000
# Items
y = 4x - 4
Insertion Sort
Merge Sort
Quick Sort


Time Elapsed - Ascending Order
Time Elapsed (ms)
100000
10000
1000
100
10
1
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort


# Key Comparisons - Descending Order
y = x² + 1x - 1
y = 0.75x² + 2.5x - 1
# Key Comparisons
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort


# Data Moves - Descending Order
y = 1.5x² + 7.5x - 1.0001
y = x² + 3x - 4
1E+11
1E+10
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
Data Moves
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort


Time Elapsed - Descending Order
y = 5E-06x² - 0.0006x + 14
y = 3E-06x² - 0.0012x + 16
100000
10000
1000
100
10
1
Time ELapsed (ms)
0
20000
40000
60000
80000
100000
120000
140000
Items
Insertion Sort
Merge Sort
Quick Sort

Q4) Interpretation

Insertion sort was $\theta(n^2)$ in random order and descending order. We can see this if we look at the equations on number of key comparisons & data moves and the time elapsed. When the array was sorted in the ascending order, insertion sort was $\theta(n)$, as we can see from the equations of plots in insertion sort. The time elapsed was unreliable in this case (sorting an array in ascending order using insertion sort). Because, the algorithm was so fast that other factors (system temperature, memory allocations etc.) started to matter, giving us unreliable data. Theoritical calculations show that number of key comparisons and data moves was related to n^2 in the worst case and average case and related to n in the best case. The reason for time elapsed plot to be like this could be the size of arrays; we might have needed a larger interval for the best case in insertion sort.

For quick sort and merge sort, the plot was not able to make an equation out of the data. What we can do is we can compare the height of the plots. In merge sort, the plot was almost in the same shape for number of key comparisons and data moves. For the time elapsed, the same problem with the insertions sort's best case was encountered. Theoritical results show that merge sort's efficiency is same ($\theta(n\log n)$) in the worst case, average case and the best case. If we compare the plots for # key comparisons and data moves in ascending order, we can see that merge sort's efficiency is worse than $\theta(n)$ but better than $\theta(n^2)$, where $\theta(n\log n)$ belongs.

In quick sort, the results actually depend if the array is sorted or not. In the plots for randomized arrays, we can see that # key comparisons and data moves are almost the same as merge sort and much more smaller than insertion sort. The time elapsed plot, again, is not that regular. However, we can see from the time elapsed plot that merge sort and quick sort are on the same page if the array is randomized. Since the merge sort was observed to be worse than $\theta(n)$ but better than $\theta(n^2)$, this would be also true for quick sort. Therefore quick sort is also in the interval where $\theta(n\log n)$ is. If the array is sorted though, things start to change. In the plots for arrays in descending order, quick sort is as efficient as insertion sort. For ascending order, quick sort is the worst despite using recursion. Both cases are known as the worst cases for quick sort, that is if our array is sorted and we choose the first element as pivot. Theoritical results

show that quick sort is $\theta(n^2)$ in the worst case, which can be seen from # key comparisons, data moves & time elapsed plots in ascending/descending order.

Finally, quick sort might not be a good choice if the array is sorted and pivot is chosen as the first element. However, the probability for this to happen is actually very small and it decreases as the # of items increase (2/n! if the items are distinct). Merge sort might seem a reliable algorithm from this experiment but it takes more space than quick sort. If we wanted to sort large arrays and we want for this to work in extreme cases, we would use merge sort. If we don't care about extreme cases, we would use quick sort. If we were on small arrays, we would use insertion sort.